COSC460 PROJECT REPORT

Department of Computer Science
University of Canterbury
New Zealand.

SYBIL - A Mechanical Theorem Prover

by P.J. Marychurch

October 1983.

No report such as this is produced by one person alone. This report would not have been without the support and encouragement of my supervisors Rod Harries and Alistair Moffat.

CONTENTS

INTRODUCTION 1

TECHNICAL INTRODUCTION 2

- First Order Logic
- Interpretaion
- Validity
- Satisfaction
- Axioms and Theorems
- Herbrand's Theorems
- Resolution
- Proving

SYBIL 11

- Parse
- Reduction
- Prove
- Tree Search Strategy

CRITICISM 20

- Performance
- Factorization
- The Problem with Axioms

CONCLUSION 23

APPENDICES 24

- A. SYBIL Input Syntax
- B. Factors in OL-deduction
- C. SYBIL Example
- D. How to Use SYBIL
- E. Abbreviated Proof
- F. Implementation

REFERENCES 34

EXTERNAL APPENDICES attached

- Prime Comoutput
- Prime Run
- Cross Reference
- Listing

INTRODUCTION

The applications of mechanical theorem proving range over many types of problems. Program analysis, number theory, state transformation problems and question answering all fall within its theoretical capabilities. The basic methods for proving theorems is very simple, but their pure application puts most problems outside of physical possibility. In response to this, refinements, strategies and heuristics must be employed to reach the solution, without violating soundness or completeness in the process. SYBIL goes some small way to achieving this, and illustrates some of the difficulties.

TECHNICAL INTRODUCTION

The following section is included to provide the terminology and background that is essential to an understanding of SYBIL. The objective of this information is not to give develop a rigorous framework or formal system, but to take the reader through a quick tutorial, to allow an appreciation of what is to come.

FIRST ORDER LOGIC

First order logic is made up of formulae which consist of terms, atoms, quantifiers and operators. They are defined as follows:

A term is defined recursively as

(i) A variable is a term

(ii) If 'f' is an n-place function symbol, and 't_1', ..., 't_n' are terms, then '$f(t_1, \ldots, t_n)$' is a term. A 0-place function is known as a constant and is written 'f'.

An atom is defined as

(i) If 'P' is an n-place predicate symbol, and 't_1', ..., 't_n' are terms, then '$P(t_1, \ldots, t_n)$' is an atom. A 0-place predicate is written 'P'.

A formula is defined recursively as

(i) An atom is a formula.

(ii) If 'F' and 'G' are formulae, then the following

combinations with the operators are also formulae:

'~ F', '(F + G)', '(F & G)',
'(F => G)', '(F <=> G)'.

(iii) If 'F' is a formula, then the following combinations with quantifiers are wffs:

'[A x] F', '[E x] F'.

Informally, parentheses may be omitted since the hierarchy prevents any ambiguous interpretation.

(highest)	quantifiers
	~
	&
	+
	=>
(lowest)	<=>

Both negated and unnegated atoms are known as literals.

INTERPRETATION

An interpretation of a formula A consists of:

(i) A non-empty domain D.

(ii) An assignment to each 0-place function (constant) of an element of D.

(iii) An assignment to each n-place (n > 0) function f, a mapping

$$f: D^n \rightarrow D$$

(iv) An assignment to each 0-place predicate of an element of { T, F }.

(v) An assignment to each n-place (n > 0) predicate P, a mapping

$$P: D^n \rightarrow \{ T, F \}$$

The above consist the statement of the domain. The following definitions are required to assign a truth value to the formula as a whole.

(i) The use by the operators of their standard mappings, with the arguments provided by the relevant formulae.

(ii) An assignment to the formula [A x] G of T iff G evaluates to T for every assignment to x from D in G, otherwise F.

(iii) An assignment to the formula [E x] G of F iff G evaluates to F for every assignment to x from D in G, otherwise T.

If the entire formula A evaluates to T, then it is said to be true in D.

VALIDITY

A formula A is valid iff it is true for every interpretation, over every domain D.

SATISFACTION

A formula A is satisfiable (unsatisfiable) iff there is some (no) interpretation, over some (any) domain, for which A evaluates to true.

AXIOMS AND THEOREMS

Proving a theorem consists of showing that, given the axioms, the theorem follows. This is equivalent to proving that the theorem is the semantic consequence of the axioms. Let us represent the axioms as A_1, A_2, ..., A_n and the theorem as T. To prove the theorem it must be shown that

$$(A_1 \ \& \ A_2 \ \& \ \ldots \ \& \ A_n) \Rightarrow T \text{ is valid}$$

(true under every interpretation).

Equivalently one can show that

$$\begin{aligned} & \sim ((A_1 \ \& \ \ldots \ \& \ A_n) \Rightarrow T) && \text{is unsatisfiable} \\ = \ & \sim (\sim(A_1 \ \& \ \ldots \ \& \ A_n) + T) && \text{is unsatisfiable} \\ = \ & (A_1 \ \& \ \ldots \ \& \ A_n \ \& \ (\sim T)) && \text{is unsatisfiable.} \end{aligned}$$

It is this final format that is used in resolution theorem proving. Matters are simplified if all axioms, and the negated theorem, are converted to clauses. This form, a conjunction of disjunctions, is the one used by proof methods based on Herbrand's theorems.

HERBRAND'S THEOREMS

There is an obvious difficulty with showing unsatisfiability. It must be shown that the formula, or set of clauses, is false under ALL interpretations over ALL domains. This is clearly a monumentous task. Herbrand however, gave a procedure to construct a domain, H, for a set of clauses S, such that, if S is false under all interpretations over this domain H, then S is unsatisfiable. H is known as the Herbrand universe.

The following from Chang and Lee [CH] defines H.

'DEFINITION:

Let H_0 be the set of constants appearing in S.

If no constant appears in S, then H_0 is to consist of a single constant, say $H_0 = \{ a \}$.

For $i = 0, 1, 2, \ldots$, let H_{i+1} be the union of H_i and the set of terms of the form $f_n(t_1, \ldots, t_n)$ for all n-place functions f_n occurring in S, where t_j, $j = 1, \ldots, n$, are members of the set H_i.

Then each H_i is called the i-level constant set of S, and H_∞ is called the Herbrand universe of S.'

[pp. 52]

eg.

Let S = { R(x) + P(y, y),
~ R(f(x)),
~ P(a, f(x)) + P(b, b) }

then

H = { a, b,
f(a), f(b),
f(f(a)), f(f(b)), ... }

RESOLUTION

Resolution is due to Robinson, which he discovered in 1965 [RO]. It is used to test the unsatisfiability of a set, S, of clauses. Applying resolution to a set of clauses produces new clauses. If, and only if, the empty clause is eventually produced by resolution, then the original set S was unsatisfiable.

This ability to show unsatisfiability is used in formulae in the format given in the previous section.

Resolution is defined in the references [CH], [LO], [RO]. Below is an informal demonstration which introduces ideas, needed in Appendix B., to show that factoring is unnecessary.

In the following example each clause C denotes a set of ground clauses G, H is the Herbrand universe of S.

$$H = \{ a, b \}$$

$$C_1 = P(x) + Q(x, b) + \sim R(y)$$
$$G_1 = \{ P(a) + Q(a, b) + \sim R(a),$$
$$P(a) + Q(a, b) + \sim R(b),$$
$$P(b) + Q(b, b) + \sim R(a),$$
$$P(b) + Q(b, b) + \sim R(b) \}$$

$$C_2 = P(a) + \sim Q(y, y)$$
$$G_2 = \{ P(a) + \sim Q(a, a),$$
$$P(a) + \sim Q(b, b) \}$$

Resolution involves finding two clauses, each with one of a pair of unifiable atoms, and exactly one of the pair negated. These literals are 'unified', converted to matching literals by the substitution of functions and variables for variables. The substitutions of a variable affect all occurrences of the variable in the clauses. A new clause is produced which is the union of the altered clauses, less the literals resolved upon. The two original clauses are termed, the parents, and the new clause, the resolvent.

For example, the resolution of C_1 and C_2 produces

$$C = P(b) + \sim R(y) + P(a)$$

$Q(x, b)$ and $\sim Q(y, y)$ were the literals resolved upon.

Resolution can also be defined by its effect on the ground

clauses of the involved clauses. The set of ground clauses of the resolvent can be seen to be

$$G = \{ g : g = g_1 + g_2 \text{ where } g_1 + L_1 \text{ member of } G_1 \text{ and } g_2 + L_2 \text{ member of } G_2 \text{ and } L_1 = \sim L_2 \}$$

$$= \{ P(b) + \sim R(a) + P(a),\ P(B) + \sim R(b) + P(a) \}$$

This is also the set found by examining C. The example uses a finite Herbrand universe, but the result holds in the general case.

The empty clause has an empty ground set.

PROVING THEOREMS

The theorem can be proved if the set of clauses can be shown to be unsatisfiable by using resolution, this is equivalent to deriving the empty clause from the input set of clauses and other resolvents.

There is an obvious strategy to achieve this end.

Let $S_0 = S$

Let $S_{i+1} = S_i \cup \{$ all the resolvents whose parents are members of $S_i \}$

for $i = 0, 1, 2, \ldots$

Continue until the empty clause is derived.

This is the level saturation method. It is extremely costly both in space and time, involving extensive duplication. Two techniques can be used to improve on this. It is on these techniques that most strategies are based. They are, restricting which clauses can act as parents, and, deleting clauses from the various sets according to some criteria.

For example, input resolution contains the restriction that one of the parents must be an input clause. This strategy, however, is not complete. Not all provable theorems can be proved with this method.

Ordered-linear resolution is defined as follows

'DEFINITION: Given a set S of ordered clauses and an ordered clause C_0 in S, an OL-deduction of C_n from S with top ordered clause C_0 is a deduction, ... , which satisfies the following conditions:

(i) For i = 0, 1, 2, ..., (n - 1), C_{i+1} is an ordered resolvent of C_i (called a centre ordered clause) against B_i (called a side ordered clause); the literal resolved upon in C_i (or an ordered factor of C_i) is the last literal.

(ii) Each B_i is either an ordered clause in S or an instance of some C_j, j < i. B_i is an instance of some C_j, j < i, if and only if C_i is a reducible ordered clause. In this case, C_{i+1} is the reduced ordered clause of C_i.

(iii) No tautology is in the deduction.'

[CH pp. 140]

At first glance, OL-resolution appears very similar to the incomplete input resolution strategy. The centre clause is only ever resolved with input clauses. But OL-resolution also resolves with previous centres by way of framed literals - resolved literals that are retained in the centre clause. Framed literals carry down the information from previous resolutions and give the algorithm the ability to detect exactly when an old centre clause, rather then an input clause, should be resolved with. This is best illustrated with an example.

Resolve the ordered centre clause

[P(x)] + R(y) ([...] denotes framing)

with the input clause

~ R(z) + ~ P(z)

The last literal of the centre clause, R(y), is the one resolved on, giving

[P(x)] + R(y) + ~ R(y) + ~ P(y)

The first of the resolved literals is framed, the second is deleted.

[P(x)] + [R(y)] + ~ P(y)

It is then noticed that the new 'last literal' can be resolved with a previous framed literal in the same clause. This resolution is equivalent to resolving with an earlier centre clause. The reduction, as it is called, gives

[P(x)] + [R(y)]

Any framed literal appearing to the right of all other literals in the chain, is deleted. This reduction gives as the final resolvent, the empty clause.

SYBIL

PARSE

One of the objectives of SYBIL is to provide a theorem prover that accepts first order logic, a theorem prover that does not require pre-transformation from first order logic to, say, clause form. The ultimate, of course, would be to accept an English description of the problem, but that scheme is a little ambitious. Instead, it was decided that SYBIL should accept a 'standard' input representing first order logic.

Input that is 'standard' satisfies two requirements.

Firstly, it should require only small changes to convert the written form of logic to a format acceptable to SYBIL. Due to restrictions of the parsing method, typographical capabilities and possible ambiguity, some changes are unavoidable. Having no change is impossible and the answer was to minimize the difficulty and reduce the number of rules that must be borne in mind when writing for SYBIL.

The second requirement for satisfactory input is that it should be readable. A person familiar with first order logic should have no difficulty in understanding the meaning of SYBIL's input, no matter how much the styles of the reader and writer differ.

Both these requirements have been met as far as possible, and practical. In addition, enhancements to normal logic style allow more information to be carried in the notation of a proof.

In Appendix A. can be found a BNF description of the syntax acceptable to SYBIL.

Major differences between SYBIL's input and most logic notations are as follows:

(i) Most formal systems strictly describe predicate, variable and function names. Here, identifiers can be used which allows a far more descriptive naming policy, with the advantage of added clarity. For example

```
[A x] ( person(x) -> mortal(x) )
```

or

```
[E simon] same_family(simon, father_of(simon))
```

Similarly it is useful in number theory to be able to use '1' as the constant representing multiplicative identity, and so on.

(ii) The normal symbols for quantifiers (rotated A and E), implication (hook) and equivalence (triple bar) are not available for typographical reasons - they do not appear in the ASCII character set. A common symbol for alternation, 'v', is not used in this way as it leads to confusion with predicate identifiers.

(iii) The solution to the problem of quantifiers was solved by representing them by the unrotated letters. This in turn led to difficulties and further restrictions were forced. The use of square brackets for, and only for, the indication of quantifier lists is a loss, but bearable.

(iv) The extra-axiomatic symbols, $ and $$ to denote individual axioms and the theorem. They are completely artificial and fulfill a role not encountered in written logic.

Although not expressed in the BNF, several semantic points should be made:

(i) All variables must be explicitly quantified. Unless this is done they are assumed to be constants.

(ii) No checking is done to ensure that all occurrences of, say, a predicate, have a parameter list of the same number of terms. In fact it would be possible to have several predicates or functions with the same name and differing number of terms, and they would be interpreted as being distinct.

(iii) Comments are denoted by a semicolon and extend to the end of a line. In addition, the rest of the line after an axiom or theorem symbol is ignored, and so can be used as a comment.

(iv) Spaces, tabs and newlines may appear anywhere except inside an identifier. Spaces are only required to separate a quantifier from its first variable.

The parsing algorithm used is a recursive descent LL(1) parse. This method has sufficient power to parse the desired input. In addition it is fast and easy to write. The parse tree constructed contains the basis of the material that will later be used in the proof proper. Most of the heap space for the initial clauses is contained in the parse tree at this stage.

Error detection and recovery is contained in the parser to provide assistance to the user.

The theorem should not be stated in its negated form. This is added by the parser.

REDUCTION

Producing clauses from the parse tree is a purely mechanical operation. The steps are well defined, although there are variances in their order and use. It is a task, however, that is tiresome and error-prone to do by hand. Much rewriting is required in successive transformations of the parse tree to its final form as a list of clauses. The existence of this section of SYBIL saves the user the ordeal.

The stages in the reduction of the tree to clause form are as follows. To illustrate the process, axioms from the prime number problem given above, will be transformed.

(i) Removal of equivalence and implication operators using the laws

```
F  <->  G   =   ( F  ->  G )  &  ( G  ->  F )
F   ->  G   =   ( ~ F )  +  G
```

```
eg.       [A x, y] (D(x, f(y)) => L(y, x))
becomes   [A x, y] (~ D(x, f(y)) + L(y, x))
```

(ii) Reduce the scope of negations to positions immediately before atoms using the laws

```
~ ( ~ F )   =   F
~ ( [A x] F[ x ] )   =   [E x] ( ~ F[ x ] )
~ ( [E x] F[ x ] )   =   [A x] ( ~ F[ x ] )
```

(De Morgan's)

```
~ ( F  +  G )   =   ( ~ F )  &  ( ~ G )
~ ( F  &  G )   =   ( ~ F )  +  ( ~ G )
```

```
eg.       ~ [A x] [E y] (P(y) & L(x, y) & ~ L(f(x), y))
becomes   [E x] [A y] (~ P(y) + ~ L(x, y) + L(f(x), y))
```

(iii) About this point, algorithms for the reduction process prescribe the renaming of variables. However SYBIL provides a unique name for each distinct variable during the parse, upon encountering that variable`s quantifier. So this step is unnecessary.

(iv) A second step that causes a more significant difference of opinion is this step, which involves the moving of quantifiers to prepare for the introduction of Skolem functions. It is an optional step, but one but one whose use can effect the difficulty of the rest of the transformation, and the proof itself.

Loveland [LO] recommends moving the quantifiers inwards, to as small a scope as possible. This may, in some cases, reduce the size of the Skolem functions produced, which, in turn, would significantly improve the proof.

Chang and Lee [CH], on the other hand, quote the same laws to move quantifiers, but apply them in reverse to Loveland. Their method increases the scope of the quantifiers, and thus the resultant Skolem functions with a net detrimental effect on the efficiency of the proof. The objective of their use of the laws is to obtain the formula in prenex normal form - a series of quantifiers to the left of the quantifier free formula (the matrix).

SYBIL does not implement this step, and so comes closer to the Loveland method. Prenex normal form is never obtained. As a result, quantifiers are left scattered throughout the tree. Their removal is left to the next step, Skolemization.

(v) Embedded quantifiers are removed, and Skolem functions introduced, by a complex tree traversal. Firstly, the tree is scanned from the root down in a breadth-first manner. At each node it is then known which universally quantified

variables have that node within scope. Upon reaching a leaf, the traversal climbs back up the tree, removing existential quantifiers as it goes. Each such quantifier removed has, by then, a quantifier-less subtree within scope. It is in this subtree that all occurrences of the existentially quantified variable are replaced by the relevant Skolem function. This down-up-down operation continues until the entire tree is free of quantifiers. Universal quantifiers are removed when their subtree becomes quantifier free.

At the termination of the traversal, all variables still remaining are assumed to be universally quantified. (The removal of their quantifiers is equivalent to moving them to the prefix.)

eg. [A x] [E y] (P(x) + (D(y, x) & P(y) & L(y, x)))
becomes P(x) + (D(g(x), x) & P(g(x)) & L(g(x), x))

(vi) The final step is straight-forward. At this point the formula consists of a conjunctive, disjunctive mix of negated and unnegated literals. The desired list of clauses is a conjunction of disjunctions, that is, conjunctive normal form. Using the laws

$$A + (B \& C) = (A + B) \& (A + C)$$
$$(B \& C) + A = (B + A) \& (C + A)$$

the formula can be converted to the normal form. The conjunctions are then dispensed with and the remaining disjunctions of literals are placed in individual clauses as lists of literals.

eg. P(x) + (D(g(x), x) & P(g(x)) & L(g(x), x)
becomes { P(x) D(g(x), x),
P(x) P(g(x)),
P(x) L(g(x), x) }

The reduction step does not affect the logical properties of the axioms and theorem. It converts a parse tree into a list of clauses, each clause a list, or a chain, of literals. It performs the role of an interface between the parser, whose output reflects the original expressions fed to SYBIL, and the prover itself, whose input is required in a strict format.

PROVE

Parsing of first order logic formulae, error detection, tree transformation, all are subsidary to the basic purpose of SYBIL, to prove formulae. As was explained in the technical introduction, Herbrand based theorem proving methods involve showing the unsatisfiability of the formula

$$A_1 \& A_2 \& \ldots \& A_n \& (\sim T)$$

This is done by demonstrating that no substitution from the Herbrand universe for the variables gives an instance that is true under some interpretation. This task is combinatorially very difficult. The NP-complete Satisfiability Problem of Cook is a subcase of theorem proving. So the designer of a theorem prover is faced with a choice, sacrifice completeness, or lose the physical capability to prove difficult theorems. It has to be admitted of both options that not all true formulae will be able to be proved to be formulae.

SYBIL's choice was to retain completeness.

The proof strategy chosen was OL-deduction of Chang and Lee [CH]. It is similar to Model Elimination of Loveland [LO]. Its algorithm has already been explained in the technical introduction.

After performing tree reduction, clauses are stored in either the axiom or theorem clause lists. this arrangement allows the implementation a variety of strategies. The requirements of the strategy actually used are not affected by the parsing or tree reduction procedures except as is noted in Appendix F.

TREE SEARCH STRATEGY

Given a centre clause, there are, generally, several possible resolutions with input clauses, and therefore several different new centre clauses. Each of these alternatives has a further group of subsequent centre clauses. A tree, in fact, of possible OL-resolutions hangs beneath each centre clause. The tree we are most concerned with hangs beneath the first centre clause. These trees contain two features which affect our choice of tree search. Scattered throughout the tree are empty clauses, the goal of the deduction. Many nodes (clauses) are identical, and so their subtrees are also identical. Some branches are of infinite length. There are terminal nodes which are not empty clauses - corresponding to centre clauses which cannot be resolved with any input clause.

An unrestricted depth first search is senseless as an otherwise simple proof may be missed if a 'deep' branch is chosen. Heuristics can suggest which branches are of this sort, but they can not ensure good choices.

Restricted depth first search does not suffer from the failing of chasing runaway branches. Similarly, breadth first search (restricted depth first with a depth of one), will not follow fruitless trails. Both allow duplicate nodes to be abandoned, although breadth first does this most efficiently. Terminal nodes offer no difficulties with either method.

The ability to find a refutation is based on the number of clauses examined. The empty clauses can be anywhere in the tree. Their discovery is as likely by searching one hundred nodes in a

subtree ten deep, as in searching the one hundred nodes closest to the root. Breadth first search is slightly easier to implement, and arbitrary decisions about depth restrictions should be avoided. SYBIL uses breadth first search.

As clauses are generated they are stored in the following structure.

```
(xxx) - (xxx) - ... - (xxx) - (xxx) - (xxx) - ... - (xxx)
head                          middle                tail
```

The middle clause is the current centre clause and the pointer moves towards the tail when all the possible resolutions have been made for that centre clause. The clause from head up to middle are past centres, and those beyond middle, up to tail, are centres to be. Each newly generated clause is compared with all the existing clauses between head and tail, and only if it is unique is it added to the list, at the tail. To reduce the time spent in making clause comparisons, only surface features are compared if that is all that is needed to differentiate two clauses. Only if the sizes match are the negations, framing and predicate names of the literals in the lists compared. If a pair of clauses is still the same then each literal pair is tested for equality. This final stage is only ever reached by identical or near identical clauses.

CRITICISM

PERFORMANCE

The proof of the existence of an infinite number of primes, is non-trivial. In proving this theorem, SYBIL produced seventy-two clauses, eight of them from the axioms and theorem. This took just over six seconds of CPU time on the Prime. With a variety of problems, SYBIL averages about ten new clauses per second initially, although this rate decreases as the number of clauses grows. It can be observed, however, that problems tend to fall into two groups, those whose solution takes less then ten seconds and those whose solution takes orders of magnitude longer. A major determinant of the length of time of a proof is the minimality of the axiom set. Redundant axioms generate a considerable number of duplicate, or unhelpful clauses. Memory capacity restricts the program to between one and two thousand clauses and by this point SYBIL is averaging only two clauses per second. This number consumes sixteen segments, or about two megabytes of memory, using around six hundred seconds of CPU time in the process. Thus the primary physical restriction on SYBIL is usually memory rather then time.

FACTORIZATION

Loveland [LO] in his Model Elimination and Chang and Lee [CH] in their OL-deduction mention factorization as one of the sources from which to derive a new clause from the centre clause. For example, Chang and Lee [CH] state

'DEFINITION : An ordered resolvent of an ordered clause C_1 against an ordered clause C_2 is any of the following binary resolvents:

1. An ordered binary resolvent of C_1 against C_2;
2. An ordered binary resolvent of C_1 against an ordered factor of C_2;
3. An ordered binary resolvent of an ordered factor of C_1 against C_2;
4. An ordered binary resolvent of an ordered factor of C_1 against an ordered factor of C_2.'

[pp. 140]

SYBIL does not use factorization. None of the soundness or completeness proofs of the above works require the use of factorization as part of the strategy. In Appendix B. is given a proof of its dispensibility since it is felt that the authors leave its position uncertain.

THE PROBLEM WITH AXIOMS

OL-deduction may arrive at an incorrect conclusion if the axioms fail to satisfy certain conditions.

If the axioms are inconsistent, OL-deduction may not be sound. In the case of inconsistency, any theorem is true, but the proof strategy may fail to find this. In fact, it may claim the opposite, that the theorem is false. For example

Axioms	p(a)	
	~ p(a)	- inconsistent/unsatisfiable
Theorem	~ p(b)	- (not yet negated)

Taking the theorem as the centre clause, no resolutions are possible, the theorem is therefore claimed to be false. But obviously the set of clauses is unsatisfiable due to the inconsistency of the two clauses.

At present SYBIL fails to take this into account. Thus, the conclusion is a dubious one if the theorem was broken into two or more clauses. A solution is to progressively deal with the clauses that make up the negated theorem. If a clause is found to be consistent with the set of input clauses, then add it to that set. (Its presence may be necessary to come to a conclusion about another of the theorem clauses). If the empty clause is derived from one of the theorem clauses, then the theorem is proved. If, after going through all the theorem clauses the empty clause has not been deduced, and the original set of axioms was satisfiable, then the theorem is disproved.

Previous theorem provers avoid this problem since they are generally given only pre-checked axioms, and the one theorem clause that leads to unsatisfiability.

CONCLUSION

At the start of this project several objectives were specified with regard to what SYBIL should achieve. The requirement of 'standard' input has been compromised due to restrictions of the equipment and, to a lesser extent, by limitations of the LL(1) parser used. The need for completeness has been met, but at the inevitable price of putting some theorems beyond the physical reach of SYBIL. Ease of use has been maintained. Its speed is quite pleasing, and additional heuristic improvements can easily be incorporated.

Like all resolution theorem provers, SYBIL suffers from a wealth of detail and no instinct. The ineffectiveness of this brute-force approach can be observed in the effect of tuning the axioms. For instance, in the prime number problem, the use of the factorial function is a mark of human input, and acts as a signpost to the theorem prover. Operating on such contrived examples SYBIL can function well. But given no signposts or redundant or ill-chosen axioms, SYBIL may be unable to find the simple but obscured solution.

Overall, SYBIL performs well for the type of prover it is, but as a device to prove theorems, it is not suitable. SYBIL still has value though, and provides a solid basis on which future ideas and implementations may be built.

APPENDIX A.

SYBIL INPUT SYNTAX

```
<SYBIL>          ::=  <axiomlist> <theorem>

<idchar>         ::=  a  |  b  | ... |  y  |  z  |
                      A  |  B  | ... |  Y  |  Z  |
                      0  |  1  | ... |  8  |  9  |  _

<identifier>     ::=  <idchar>  |  <idchar> <identifier>
<idlist>         ::=     |  , <identifier> <idlist>

<term>           ::=  <identifier> <oppara>
<termlist>       ::=     |  , <term> <termlist>

<oppara>         ::=     |  ( <term> <termlist> )

<predicate>      ::=  <identifier> <oppara>

<quantifier>     ::=  E  |  A
<quanpart>       ::=  [ <quantifier> <identifier> <idlist> ]

<equivsym>       ::=  <->  |  <=>
<implsym>        ::=   ->  |  =>
<orsym>          ::=  +  |  |  |  \/
<andsym>         ::=  .  |  *  |  /\  |  &  |  ^
<notsym>         ::=  ~  |  -

<formula>        ::=  <form2a> <form1b>
<form1b>         ::=     |  <equivsym> <form2a> <form1b>
<form2a>         ::=  <form3a> <form2b>
<form2b>         ::=     |  <implsym> <form3a> <form2b>
```

```
<form3a>        ::=  <form4a> <form3b>
<form3b>        ::=     |  <orsym> < form4a> <form3b>
<form4a>        ::=  <form5a> <form4b>
<form4b>        ::=     |  <andsym> <form5a> <form4b>
<form5a>        ::=  <negsym> <form5a>  |  <form6a>
<form6a>        ::=  <predicate> |
                     ( <formula> )  |
                     <quanpart> <form6a>

<axiom>         ::=  $ <formula>
<axiomlist>     ::=     |  <axiom> <axiomlist>

<theorem>       ::=  $$ <formula>
```

APPENDIX B.

FACTORS IN OL-DEDUCTION

Theorem:

Let C be a factorable ordered centre clause. If there is an OL-refutation from the factor C' of C, then there is an OL-refutation from C that does not involve factoring.

Proof: (sketch)

Let G be the set of ordered ground clauses of C, and G' be the similar set for C'. OL-resolutions strip off the right end literals from clauses. Since there is an OL-refutation from a ground clause, say, D' in G', then there is a series of resolutions to strip off all the literals in this clause D until the empty clause is reached. Corresponding to D' of G' is an ordered ground clause D of G. These are two clauses are identical except that D has an extra literal at the right hand end. However this extra literal, the one that is factored on, has an identical copy, the other literal factored on, to the left. Since this left-most literal can be removed by an OL-deduction then the right-most one can be also. The other literals can be removed as they are in the OL-refutation from C', until the empty clause is derived. Thus factoring is unnecessary in OL-reolution.

APPENDIX C.

SYBIL EXAMPLE

```
;
; There exist infinitely many primes
;
;       L(x, y)    'x < y'
;       D(x, y)    'x divides y'
;       P(x)       'x is a prime'
;       f(x)       'x! + 1'
;
$
[A x] (~ L(x, x))
        ; No number is less than itself
$
~ [E x, y] (L(x, y) & L(y, x))
        ; No two numbers can be less then each other
$
[A x, y] (D(x, f(y)) => L(y, x))
        ; If x divides (y! + 1) then y is less than x
        ; This is an axiom that could well be the subject
        ;       of a proof itself
$
[A x] L(x, f(x))
        ; x < (x! + 1)
$
[A x] [E a] (P(x) + (D(a, x) . P(a) . L(a, x)))
        ; Either x is a prime, or there is a number less
        ;       then x, that divides x, and is itself a prime
$$
[A x] [E y] (P(y) & L(x, y) & ~ L(f(x), y))
        ; For every number, x, there is a prime y such that
        ;       x < y <= (x! + 1)
```

APPENDIX D.

HOW TO USE SYBIL

SYBIL <input file> [options]

The input file should conform to accepted syntax. Errors are flagged.

Options:

-L, -LIST, -LISTING <list file> : Specify a file to send the listing to. File <input file>.LIST is used as a default. There is no provision for turning the listing off. An option such as -LIST ' ' will send the listing to the screen.

-D, -DUMP, -DIAG, -DIAGNOSTIC <dump file> : Specify a file to send a dump of the parsing and tree reduction of the input. File <dump file>.DUMP is used as a default. This switch is normally off and is of interest only for debugging SYBIL, not for tracing the input. Some additional diagnostics come to the screen. The file produced can be very large.

APPENDIX E.

ABBREVIATED PROOF

```
Clause 1   given   Size : 1
    ~l(x, x)

Clause 2   given   Size : 2
    ~l(x, y)
    ~l(y, x)

Clause 3   given   Size : 2
    ~d(x, f(y))
     l(y, x)

Clause 4   given   Size : 1
     l(x, f(x))

Clause 5   given   Size : 2
     d(f$5(x), x)
     p(x)

Clause 7   from 5   Size : 2
     p(f$5(x))
     p(x)

Clause 8   from 7   Size : 2
     l(f$5(x), x)
     p(x)

Clause 6   given   Size : 3
    ~p(x)
    ~l(f$6, x)
     l(f(f$6), x)
```

```
*****************************************


Clause 9    from 6    Size : 3
     ~p(x)
     ~l(f$6, x)
      l(f(f$6), x)

Clause 10    from 9    and 5    Size : 3
      d(f$5(x), x)
    [~p(x)]
     ~l(f$6, x)
      l(f(f$6), x)

Clause 11    from 9    and 7    Size : 3
      p(x)
    [~p(f$5(x))]
     ~l(f$6, f$5(x))
      l(f(f$6), f$5(x))

            .
            .
            .

    (removed clauses)

            .
            .
            .

Clause 71    from 60    and 6    Size : 3
     ~l(f$6, f(f$6))
      l(f(f$6), f(f$6))
```

```
   [ p(f(f$6))]
   [~d(f$5(f(f$6)), f(f$6))]
   [~l(f$6, f$5(f(f$6)))]
     l(f(f$6), f$5(f(f$6)))

Clause 72    from 67    and 1    Size : 0
Empty clause

Theorem proved

*******************************************

Proof is as follows ...


Clause 72    from 67    and 1    Size : 0
Empty clause

Clause 67    from 55    and 4    Size : 1
     l(f(f$6), f(f$6))

Clause 55    from 41    and 8    Size : 2
    ~l(f$6, f(f$6))
     l(f(f$6), f(f$6))

Clause 41    from 28    and 2    Size : 3
    ~l(f$5(f(f$6)), f(f$6))
   [ l(f(f$6), f$5(f(f$6)))]
   [ l(f$6, f$5(f(f$6)))]
   [ d(f$5(f(f$6)), f(f$6))]
   [~p(f(f$6))]
    ~l(f$6, f(f$6))
     l(f(f$6), f(f$6))
```

```
Clause 28   from 19   and 7   Size : 3
     l(f(f$6), f$5(f(f$6)))
   [ l(f$6, f$5(f(f$6)))]
   [ d(f$5(f(f$6)), f(f$6))]
   [~p(f(f$6))]
    ~l(f$6, f(f$6))
     l(f(f$6), f(f$6))

Clause 19   from 14   and 6   Size : 4
    ~p(f$5(f(f$6)))
     l(f(f$6), f$5(f(f$6)))
   [ l(f$6, f$5(f(f$6)))]
   [ d(f$5(f(f$6)), f(f$6))]
   [~p(f(f$6))]
    ~l(f$6, f(f$6))
     l(f(f$6), f(f$6))

Clause 14   from 10   and 3   Size : 3
     l(x, f$5(f(x)))
   [ d(f$5(f(x)), f(x))]
   [~p(f(x))]
    ~l(f$6, f(x))
     l(f(f$6), f(x))

Clause 10   from 9   and 5   Size : 3
     d(f$5(x), x)
   [~p(x)]
    ~l(f$6, x)
     l(f(f$6), x)
```

APPENDIX F.

IMPLEMENTATION

SYBIL is written in Sheffield Pascal on the University of Canterbury Prlme. It consists of a large number of source files linked together. For the initial user interface it uses a CPL file. Several non-standard features are used: otherwise; halt; mill; date; and the commented use of cand and cor.

A modular, top-down structure was intended, but the Sheffield Pascal Prlme implementation of external procedures prevented this. Thus the same data declarations are global to the entire program. It was found not to be possible to have some variables locally global to, say, the parser and others locally global to the prover.

Accompanying this report is the program listings for SYBIL, an index of externally accessible routines, and the complete output of SYBIL's proof of the prime number problem.

REFERENCES

[CH] 'Symbolic Logic and Mechanical Theorem Proving.'
Chin-Liang Chang and Richard Char-Tung Lee,
1973, Academic Press.

[LO] 'Automated Theorem Proving: A Logical Basis.'
Donald E. Loveland,
1978, North Holland.

[RO] 'A machine-oriented logic based on the resolution principle.'
J. A. Robinson,
1965, J. Assoc. Comput. Mach., 12, pp 23-41.